Double-click (or enter) to edit

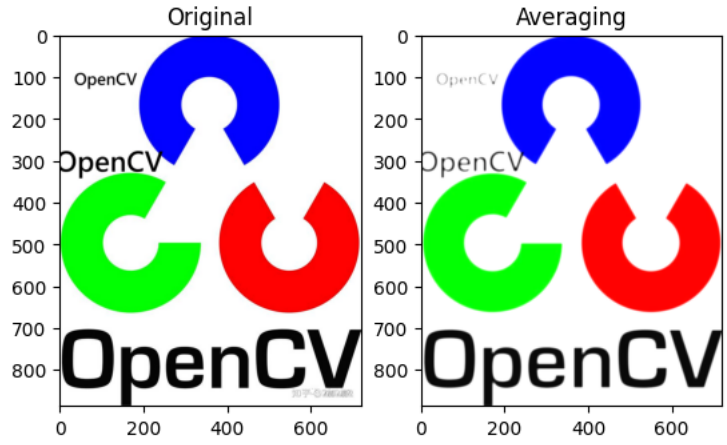
Image enhancement

It plays an important role in improving image quality in the field of image processing, which is achieved by highlighting useful information and suppressing redundant information in the image.

It is the process of adjusting digital images so that the results are more suitable for display or further image analysis. For example, you can remove noise, sharpen, or brighten an image, making it easier to identify key features.

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('Images/opencv_logo1.png')
#assert img is not None, "file could not be read, check with os.path.exists()"
kernel = np.ones((7,7),np.float32)/25
dst = cv.filter2D(img,-1,kernel)
```

```
plt.subplot(121),plt.imshow(img),plt.title('Original')
#plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(dst),plt.title('Averaging')
#plt.xticks([]), plt.yticks([])
plt.show()
cv.imwrite('Images/opencv_blurred.png',dst)
```




```
True
```

Image Blurring (Image Smoothing)

Image blurring is achieved by convolving the image with a low-pass filter kernel. It is useful for removing noise. It actually removes high frequency content (eg: noise, edges) from the image. So edges are blurred a little bit in this operation (there are also blurring techniques which don't blur the edges). OpenCV provides four main types of blurring techniques.

1. Averaging
2. Gaussian Blurring
3. Median Blurring
4. Bilateral Filtering

1. Averaging

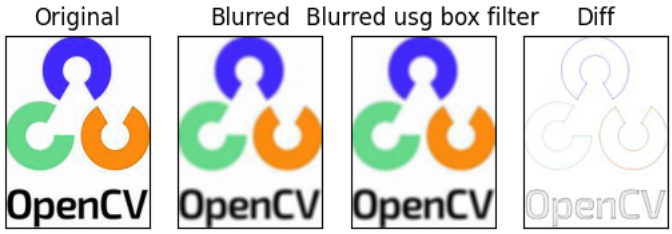
This is done by convolving an image with a normalized box filter. It simply takes the average of all the pixels under the kernel area and replaces the central element. This is done by the function cv.blur() or cv.boxFilter(). Check the docs for more details about the kernel. We should specify the width and height of the kernel. A 3x3 normalized box filter would look like the below:

image.png

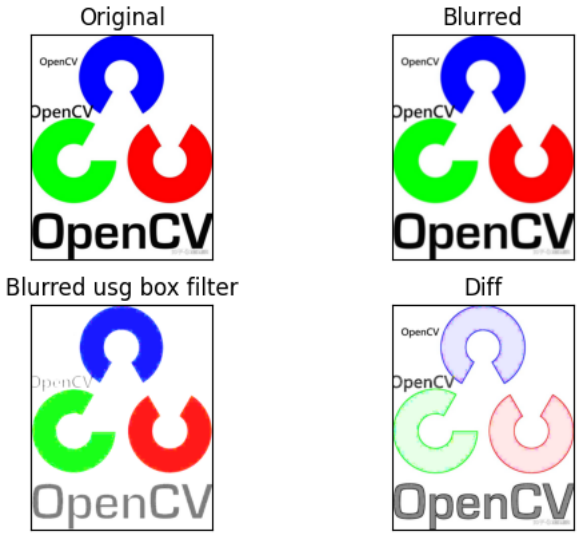
Note

If you don't want to use a normalized box filter, use cv.boxFilter(). Pass an argument normalize=False to the function. Check a sample demo below with a kernel of 5x5 size:

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt

img = cv.imread('Images/opencv_logo.png')
assert img is not None, "file could not be read, check with os.path.exists()"
blur = cv.blur(img,(5,5))

dst=cv.boxFilter(img, -1, (5,5), normalize = True)
diff=cv.add(img,-dst)

plt.subplot(141),plt.imshow(img),plt.title('Original')
plt.xticks([]), plt.yticks([])
plt.subplot(142),plt.imshow(blur),plt.title('Blurred')
plt.xticks([]), plt.yticks([])
plt.subplot(143),plt.imshow(dst),plt.title('Blurred usg box filter')
plt.xticks([]), plt.yticks([])
plt.subplot(144),plt.imshow(diff),plt.title('Diff')
plt.xticks([]), plt.yticks([])

plt.show()
```




```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt

img = cv.imread('Images/opencv_logo1.png')
assert img is not None, "file could not be read, check with os.path.exists()"
blur = cv.blur(img,(5,5))

dst=cv.boxFilter(img, -1, (5,5), normalize = False)
diff=cv.add(img,-dst)
#plt.figure(figsize=(6,5))
plt.subplot(221),plt.imshow(img),plt.title('Original')
plt.xticks([]), plt.yticks([])
plt.subplot(222),plt.imshow(blur),plt.title('Blurred')
plt.xticks([]), plt.yticks([])
plt.subplot(223),plt.imshow(dst),plt.title('Blurred usg box filter')
plt.xticks([]), plt.yticks([])
plt.subplot(224),plt.imshow(diff),plt.title('Diff')
plt.xticks([]), plt.yticks([])

plt.show()
```



2. Gaussian Blurring

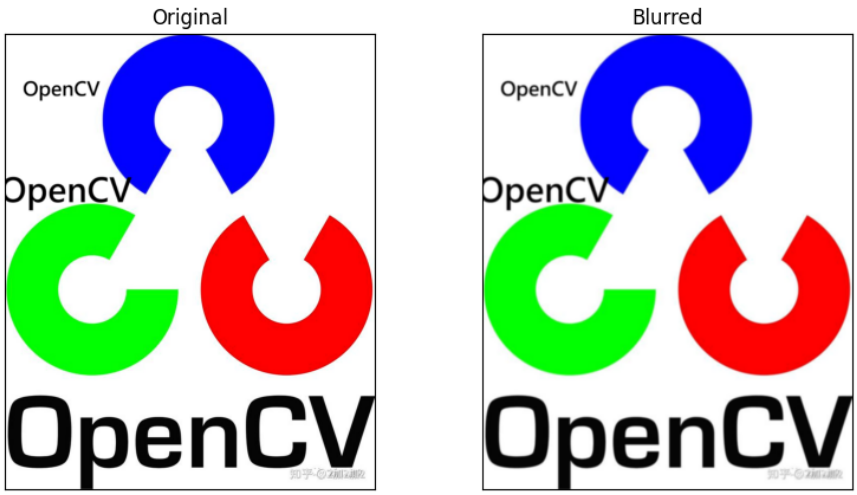
In this method, instead of a box filter, a Gaussian kernel is used. It is done with the function, cv.GaussianBlur(). We should specify the width and height of the kernel which should be positive and odd. We also should specify the standard deviation in the X and Y directions, sigmaX and sigmaY respectively. If only sigmaX is specified, sigmaY is taken as the same as sigmaX. If both are given as zeros, they are calculated from the kernel size. Gaussian blurring is highly effective in removing Gaussian noise from an image.

If you want, you can create a Gaussian kernel with the function, cv.getGaussianKernel().

The above code can be modified for Gaussian blurring:

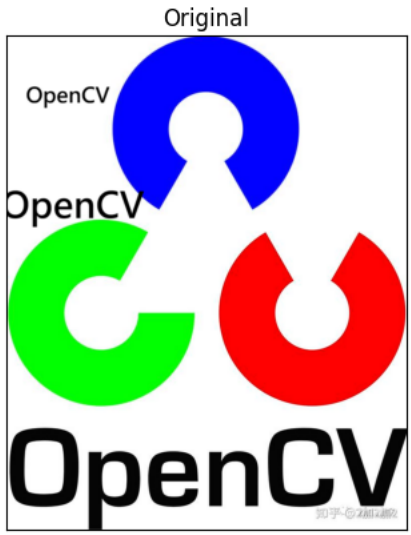
```
blur = cv.GaussianBlur(img,(5,5),0)
```

```
plt.figure(figsize=(20,5))
plt.subplot(141),plt.imshow(img),plt.title('Original')
plt.xticks([]), plt.yticks([])
plt.subplot(142),plt.imshow(blur),plt.title('Blurred')
plt.xticks([]), plt.yticks([])
plt.show()
```



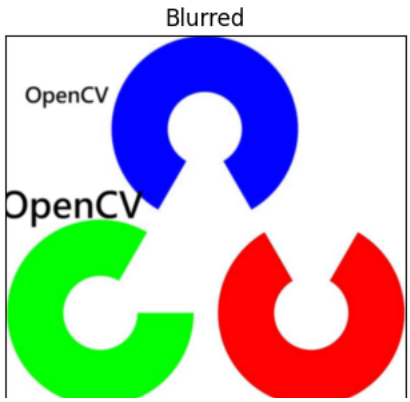

```
plt.xticks([]), plt.yticks([])
plt.imshow(img),plt.title('Original')
```

```
(<matplotlib.image.AxesImage at 0x2217a861290>, Text(0.5, 1.0, 'Original'))
```




```
plt.xticks([]), plt.yticks([])
plt.imshow(blur),plt.title('Blurred')
```

```
(<matplotlib.image.AxesImage at 0x22181ede2d0>, Text(0.5, 1.0, 'Blurred'))
```

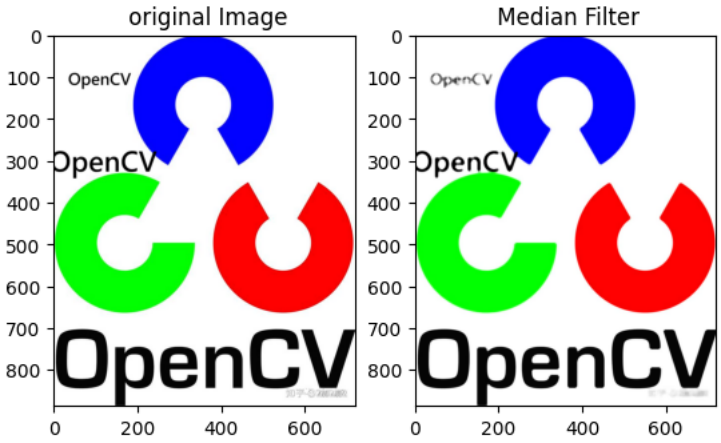


3. Median Blurring

Here, the function cv.medianBlur() takes the median of all the pixels under the kernel area and the central element is replaced with this median value. This is highly effective against salt-and-pepper noise in an image. Interestingly, in the above filters, the central element is a newly calculated value which may be a pixel value in the image or a new value. But in median blurring, the central element is always replaced by some pixel value in the image. It reduces the noise effectively. Its kernel size should be a positive odd integer.

In this demo, I added a 50% noise to our original image and applied median blurring. Check the result:

```
median = cv.medianBlur(img,9)

plt.subplot(121), plt.imshow(img), plt.title(" original Image")
plt.subplot(122), plt.imshow(median), plt.title(" Median Filter")
plt.show()
```



4. Bilateral Filtering

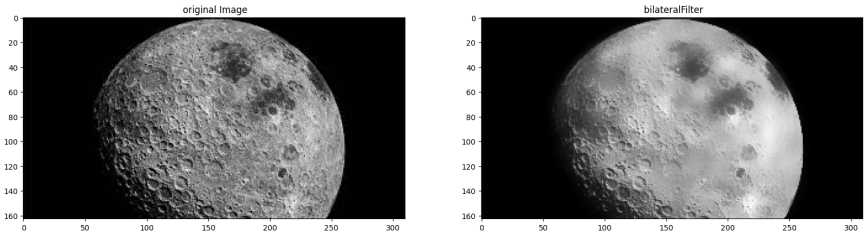
cv.bilateralFilter() is highly effective in noise removal while keeping edges sharp. But the operation is slower compared to other filters. We already saw that a Gaussian filter takes the neighbourhood around the pixel and finds its Gaussian weighted average. This Gaussian filter is a function of space alone, that is, nearby pixels are considered while filtering. It doesn't consider whether pixels have almost the same intensity. It doesn't consider whether a pixel is an edge pixel or not. So it blurs the edges also, which we don't want to do.

Bilateral filtering also takes a Gaussian filter in space, but one more Gaussian filter which is a function of pixel difference. The Gaussian function of space makes sure that only nearby pixels are considered for blurring, while the Gaussian function of intensity difference makes sure that only those pixels with similar intensities to the central pixel are considered for blurring. So it preserves the edges since pixels at edges will have large intensity variation.

The below sample shows use of a bilateral filter (For details on arguments, visit docs).

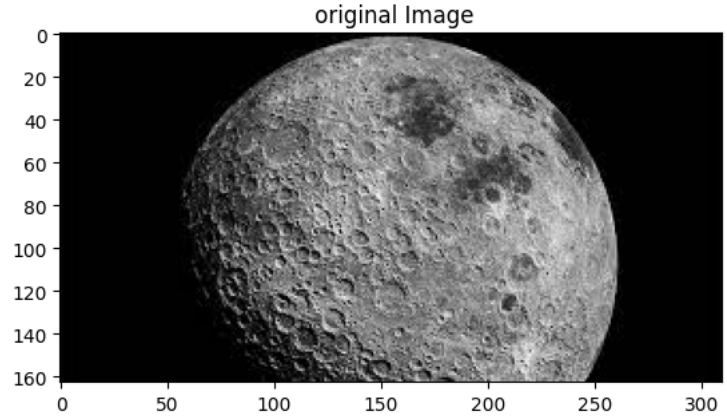
```
import cv2 as cv
import matplotlib.pyplot as plt
img=cv.imread("Images/moon1.jpeg",0)

blur = cv.bilateralFilter(img,15,50,50)
plt.figure(figsize=(20,5))
plt.subplot(121), plt.imshow(img, cmap="gray"), plt.title(" original Image")
```

```
plt.subplot(122), plt.imshow(blur, cmap="gray"), plt.title(" bilateralFilter")
plt.show()
```




```
plt.imshow(img, cmap="gray"), plt.title(" original Image")
```

```
(<matplotlib.image.AxesImage at 0x2218133ce90>,
 Text(0.5, 1.0, ' original Image'))
```




```
plt.imshow(blur, cmap="gray"), plt.title(" Bilateral Filter")
```

```
(<matplotlib.image.AxesImage at 0x2217c374e90>,
 Text(0.5, 1.0, ' Bilateral Filter'))
```

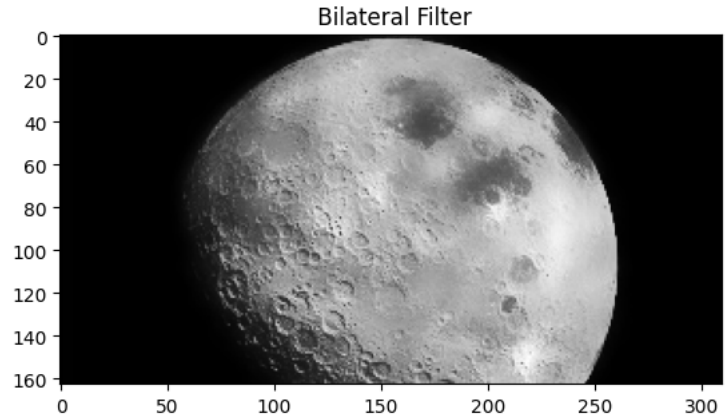


Image gradients, edges etc

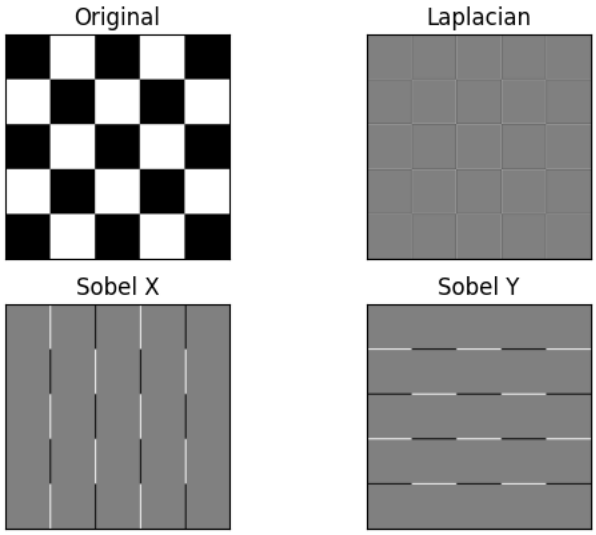
We will see following functions : cv.Sobel(), cv.Scharr(), cv.Laplacian() etc

Theory

OpenCV provides three types of gradient filters or High-pass filters, Sobel, Scharr and Laplacian. We will see each one of them.

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img_path = 'Images/dave.jpg'
img_path = 'Images/checkerboard5x5.png'
```

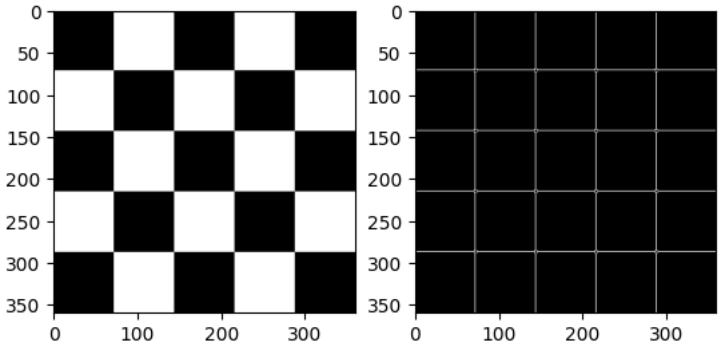
```
img = cv.imread(img_path, cv.IMREAD_GRAYSCALE)
assert img is not None, "file could not be read, check with os.path.exists()"
laplacian = cv.Laplacian(img,cv.CV_64F)
sobelx = cv.Sobel(img,cv.CV_64F,1,0,ksize=3)
sobely = cv.Sobel(img,cv.CV_64F,0,1,ksize=3)
plt.subplot(2,2,1),plt.imshow(img,cmap = 'gray')
plt.title('Original'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,2),plt.imshow(laplacian,cmap = 'gray')
plt.title('Laplacian'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,3),plt.imshow(sobelx,cmap = 'gray')
plt.title('Sobel X'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,4),plt.imshow(sobely,cmap = 'gray')
plt.title('Sobel Y'), plt.xticks([]), plt.yticks([])
plt.show()
```




```
img_path = 'Images/checkerboard5x5.png'
#Reading the image
image = cv.imread(img_path)
(H, W) = image.shape[:2]
# convert the image to grayscale
gray = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
# blur the image
blurred = cv.GaussianBlur(gray, (5, 5), 0)
# Perform the canny operator
canny = cv.Canny(blurred, 6, 50)
#Let’s see the output of the canny edge detector
```

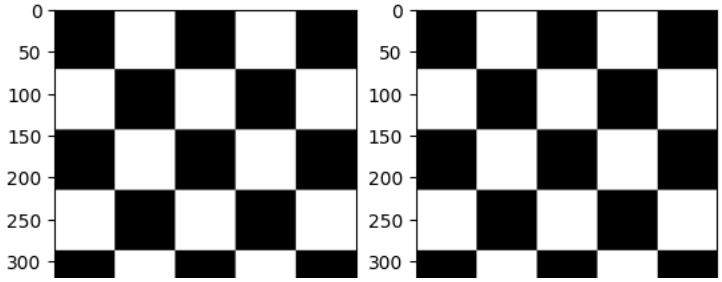
```
plt.subplot(121), plt.imshow(gray,cmap='gray')
plt.subplot(122), plt.imshow(canny,cmap='gray')
```

```
(<Axes: >, <matplotlib.image.AxesImage at 0x22181c66790>)
```




```
plt.subplot(121), plt.imshow(gray,cmap='gray')
plt.subplot(122), plt.imshow(gray,cmap='gray')
```

```
(<Axes: >, <matplotlib.image.AxesImage at 0x2217c25b690>)
```




```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('Images/messi5.jpg', cv.IMREAD_GRAYSCALE)
assert img is not None, "file could not be read, check with os.path.exists()"
edges = cv.Canny(img,100,200)
plt.subplot(121),plt.imshow(img,cmap = 'gray')
plt.title('Original Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(edges,cmap = 'gray')
plt.title('Edge Image'), plt.xticks([]), plt.yticks([])
plt.show()
```